CSC104, Assignment 3, Winter 2006
Sample solution

Danny Heap

Finger exercises

1. Create directory *A3*, make it your working directory, and create a file called *journalA3* in it. This will be where your record the joys and frustrations of working through this assignment, plus how you go about solving (or not) the exercises, and observations or investigations that occur to you on the way.

   While still in directory *A3*, start up DrScheme, by typing *drscheme*. Once you have a DrScheme window, you can set a comfortable font under *Edit/preferences*, and set the level of the Scheme language to "Intermediate student with lambda" by going to the *Language/Choose Language* menu, and looking under "How to Design Programs." The idea of selecting a language level is to make enough of Scheme available to allow you to do some interesting things, but to turn off some other features that might get you into trouble. Of course, you are free to set the language level to whatever you decide, so that you can get into as much, or little, trouble as you choose.

   While under the *Language* menu, switch to the *Teachpack* submenu, and select *htdp*. You will see a number of entries with an ".ss" extension, and you should double-click *draw.ss*. This will allow your to use some drawing commands in your programs. Although I'll mention the necessary commands from *draw.ss*, if you're curious you can look under *Help/Help Desk/Teachpacks*.

   Solution: I set up direction *A3* and started *journalA3* similarly to the first two assignments. I opened up Drscheme, and although I was able to choose the *Intermediate Student with Lambda* level, I couldn't seem to get the *draw.ss* teachpack installed. After fussing with this forever, I tried another computer, which seem to have a more compliant mouse, and I was able to put in the teachpack.

   The help listed on the teachpacks seemed terse and uninformative.

2. The bottom part of your DrScheme is an interactions pane, where you can experiment with Scheme expressions. Arithmetic uses some familiar operators such as + - * and / together with a couple of probably-unfamiliar features: expressions are wrapped in parentheses, and operators come before the things they operate on (operands). Try typing the following in the DrScheme interactions pane, and record your experience and explanations or conjectures in *journalA3* (if you get a message about the interactions pane not being synchronized with the definitions pane, click the *run* button and continue):

```
> (+ 2 3)
> (+ 1 2 3 4)
> (* 2 3)
> (* 2 3 4)
```

```
> (/ 3 5)
> (* (/ 3 5) (/ 5 3))
```

SOLUTION: Scheme appears to perform the usual arithmetic operations in an unusual way: the operator comes before the things it operates on. This seems pretty awkward, except in the case of *(+ 1 2 3 4)* it seems to save keystrokes (that adds up all the numbers that follow the "+" sign). The general rule seems to be: *(operator number1 number2)* means *number1 operator number2*. The order is important with subtraction and division: *(/ 3 5)* means 3 divided by 5, not 5 divided by 3. The parentheses tell me the scope of things to operate on, and I can embed subexpressions, for example: *(* (/ 3 5) (/ 5 3))* means to first evaluate the inner expressions (so 3/5 and 5/3), and then multiply them (producing 1). The division operator seems to produce an exact floating point result. For example, *(/ 5 3)* produces a repeating decimal, and indicates which portion repeats.

3. As well as numbers, Scheme has lists. Various types of objects (including lists themselves) can be members of lists, but numbers are probably the most concrete sort of object to begin with. Here's how you can create a list of the first five positive integers:

   ```
   (list 1 2 3 4 5)
   ```

   This is exactly the same as:

   ```
   '(1 2 3 4 5)
   ```

   Scheme provides some tools for manipulating lists. Try out, and then explain, the following:

   ```
   (first (list 1 2 3 4 5))
   (second (list 1 2 3 4 5))
   (eighth (list 1 2 3 4 5))
   (rest (list 1 2 3 4 5))
   (rest (list 6 7 8 9 10))
   (reverse (list 1 2 3 4 5))
   (reverse (reverse (list 1 2 3 4 5)))
   (length (list 1 2 3 4 5))
   (append (list 1 2 3 4) (list 5 6 7 8))
   ```

   SOLUTION: The word *list* seems to have a special meaning to Scheme, so that if it is combined with a sequence of numbers I can use other special words such as *first, second, rest, reverse* on them. It's not clear to me why I'd use it instead of the '(1 2 3 4) version, but perhaps that will come later. The word *first* extracts the first element of a list, *second* extracts the second, and so on. You have a problem if you extract the eighth element of a list that doesn't have eight elements, so that command generated an error. The word *rest* returns a list with the first element removed (the *rest* of the list, I guess). The word *reverse* does what it says: it reverses a list, and if you reverse a list twice, you get the original list back. *length* gives you the length of a list, and *append* pastes two lists together. I tried out *(length (list))*, and got "0", so that list must be empty. I also found that *(append '() (list 1 2 3))* gives me *(list 1 2 3)*, so you leave a list unchanged by appending an empty list. I tried out *(eleventh 1 2 3 4 5 6 7 8 9 10 11)*, but the word *eleventh* doesn't seem to be familiar to Scheme. How do words get their special meaning in Scheme?

4. Scheme can tell *true* from *false*, combine them with *and* and *or*. Try out the following, and record your observations and explanations:

```
> (> 3 5)
> (< 3 5)
> (positive? 5)
> (positive? -3)
> (and (< 3 5) (positive? -3))
> (or (< 3 5) (positive? -3))
```

SOLUTION: The "operator before the things it operates on" pattern continues with Scheme. If I want to test whether $3 > 5$, I put the $>$ at the beginning of my Scheme expression, and I get *false* for my trouble (which is, so to speak, true). I can also test whether 3 is less than 5 by using the $<$ operator instead. Scheme recognizes *positive?* as a question, and gives a true/false answer, depending on whether the number following it is positive or not.

The ordinary words *and* and *or* have special meaning to Scheme. If I have *and* followed by a couple of things, one of which is false, I get *false*. I tried following it with two true things, and it produces *true*. Two false things produce *false* again. Some the *and* of two things is true exactly when they both are true. I tried a similar set of tests on the special word *or*, and it seems as though the *or* of two things is true unless they are both false.

By the way, the "things" in the previous paragraph have to be expressions that can only be true or false. I tried plugging in numbers, and Scheme complained.

Scheme can choose different actions, based on whether some expression is *true* or *false*, using the special keywords *if* and *cond*. Experiment with the following until you can explain it (note that strings are denoted similarly to what you've seen in Python):

```
> (if (> 5 3) "five is more than three" "five is not more than three")
> (if (< 5 3) "five is less than three" "three is not less than three")
> (cond ((< 5 3) "five is less than three")
        ((> 5 3) "five is greater than three")
        (else "five and three are equal"))
```

SOLUTION: The general pattern here seems to be *(if (true-or-false-thing) do-this do-that)*. When the true-or-false-thing is true, we do-this, otherwise we do-that. So far as Scheme is concerned, strings such as "five is more than three" are just objects to produce, depending on whether the true-or-false-thing is true or false. I tried writing nonsense for both strings, and Scheme would produce the first one when the true-or-false-things is true, and the second one otherwise.

The next pattern seems to be (cond (((T/F?) do1) ((T/F?) do2) ...)), where if one of the (T/F?) is true, you do the corresponding do#. I wonder what happens if more than one of the T/F? is true? I tried this out, and it just returns the do# corresponding to the first thing that's true.

The special keyword *define* binds the first expression that follows it to the second expression that follows it, so you can use the first expression as a name to call the second. Try out the following example, and chat with a TA or an instructor until you can record some sort of explanation:

```
> (define (maximum n1 n2)
    (if (> n1 n2) n1
        n2))
```

```
> (maximum 3 5)
> (maximum 5 3)
```

SOLUTION: It seemed odd to be defining something as obvious as maximum, so I tried *(maximum 2 3)*, and Scheme scolded me that maximum wasn't yet defined.

I tried typing the given example, and suddenly maximum is defined: *(maximum 2 3)* produces 3, as does *(maximum 3 2)*, so the definition I created actually achieved something. I tried to trace out how it works. When a user types *(maximum 3 2)*, the definition substitutes 3 for n1 and 2 for n2, and then checks whether n1 (or 3) is greater than n2 (or 2). This is true, so n1 (or 3) is produced. Doing things the other way around, if the user types *(maximum 2 3)*, the definition substitutes 2 for n1 and 3 for n2, then checks whether n1 is greater than n2 (now this is false), so it produces n2 (or 3). This worked so well that I tried *(maximum 3 4 5)*, but Scheme scolded me that I had provided 3 arguments instead of 2 (and I wasn't feeling particularly argumentative). It looks as though I can break things down and trace through why they work.

5. The semicolon tells Scheme to ignore the remainder of a line it is on, and is thus useful for comments. In the next few exercises I will provide comments that describe the function that you are to try to create. Included in the comments are examples of how the function behaves.

   Your first task is to define the function *lcycle*, corresponding to the passage beginning with semicolons below. In your definitions pane, type a Scheme expression that defines *lcycle*. HINT: you will need to dream up a name for the parameter that refers to your list, just as I had to dream up n1 and n2 in the last exercise. You can't use the parameter name *list*, because Scheme already uses that for other things.

   You should read the comments very closely, plus refer back to the last few exercises, to guide you in writing *lcycle*. The comment on the first line indicates that *lcycle* takes a list as input, and produces a list as output. You should type your definition in the top (definition) pane (include the comments), and then test it by clicking the *run* button, and trying the new command out in the bottom (interactions) pane of DrScheme. Record your observations and explanations in *journalA3*. When you believe you have it working, save your file as *sneeze.scm* by using the *File/Save Definitions As* menu.

```
;; lcycle : list -> list
;; To produce a version of the list with the first element moved to the end.
;; example 1: (lcycle '(1 2)) produces '(2 1)
;; example 2: (lcycle '(1)) produces '(1)
;; example 3: (lcycle '(1 2 3)) produces '(2 3 1)
```

   SOLUTION: I began by imitating the last exercise: *(define (lcycle n1) ...)*. I actually tried running this, *(lcycle '(1 2))*, but Scheme complained that ... is an undefined identifier. So, assuming that the user types in something like *(lcycle '(1 2))*, the list they give me will be attached to n1, so I want to take the first element and append it to the end. I tried replacing my ... with *(append (rest n1) (first n1))*, but that produced an error, since (first n1) isn't a list. I fixed that up as *(list (first n1))* (a 1-element list), so I got a working version: *(define (lcycle n1) (append (rest n1) (list (first n1))))*, and it worked on all the given examples. I tried it on an empty list, and it broke (because there is no first and no rest), so I guess I can use some sort of "if" clause to fix that, but perhaps I'll come back to that later.

6. Keep working in the definitions pane (and saving your work occasionally by clicking the *Save* button). Don't erase or replace your definition of *lcycle*. There is another passage of comments below describing

a similar function *rcycle*. You could create *rcycle* from scratch, or perhaps think of how to create it using *lcycle* and *reverse*. Record your observations and explanations in *journalA3*.

```
;; rcycle : list -> list
;; To produce a version of the list with the last element moved to the beginning.
;; example 1: (rcycle '(1)) produces '(1)
;; example 2: (rcycle '(1 2)) produces '(2 1)
;; example 3: (rcycle '(1 2 3)) produces '(3 1 2)
```

SOLUTION: This looks so similar to lcycle, that I think it should almost write itself — it's the same operations, just working from the other end of the list. I first started cutting and pasting things from lcycle, and then I realized that I could use *reverse* twice: *(define (rcycle n1) (reverse (lcycle (reverse n1))))*. I think the name n1 isn't very explanatory, I should use list (no, that doesn't work, Scheme complains), or perhaps *lst* (thanks Sheila). Once again, the given examples (and longer lists) all worked as expected, but an empty list is a problem.

7. The functions you're working on are components of a program that will simulate and display the progress of a respiratory infection through a population. We use a positive number to indicate an individual in our population who is infected. Your next job is to define *sick-left* to decide whether the left-most number in a triple (list of three) is positive. Keep working in the definitions pane where your previous two definitions are, test your work by clicking *Run* and then trying things out in the interactions pane, and save your work by clicking the *Save* button. As always, observations, explanations, and conjectures go in *journalA3*

```
;; sick-left : list -> Whether left-most number of list is positive.
;; To decide whether the left-most number of list is positive.
;; example 1: (sick-left '(5 3 1)) produces true.
;; example 2: (sick-left '(-5 3 1)) produces false.
```

SOLUTION: I know how all the parts work, so I can paste them together easily. I extract the first element using *first*, and then figure out whether it's positive using *positive?*. So that's *(define (sick-left lst) (positive? (first lst)))*. It works on all the given examples, and the procedure doesn't even make sense on an empty list (so I didn't try it).

8. Now you should define the similar function *sick-right*. Once again, you could define this from scratch, or you could use *sick-left* and *reverse*. Record your observations and explanations. Save your definitions by clicking the *Save* button, and record the process of solving the exercise in *journalA3*.

```
;; sick-right : list -> Whether the right-most number of list is positive.
;; To decide whether the right-most number of list is positive.
;; example 1: (sick-right '(1 3 5)) produces true.
;; example 2: (sick-right '(1 -3 -5)) produces false.
```

SOLUTION: I can use the reverse trick from rcycle, since the right-most member of a list is positive exactly when the left-most member of that list reversed is positive: *(define (sick-right lst) (sick-left (reverse lst)))*. Once again, I tried it out on the given examples, and it checks out.

9. Before attempting the last (tenth) exercise, you need to understand a subtle, useful, and beautiful technique called RECURSION. I will discuss this in class, but here is an example you should experiment with until you can record an explanation.

In the open definitions pane, type the definition of *GCD* below, followed by the given example. Then click the *Step* button, and then (again) the *Step>* button on the *Stepper* window that pops up. You can step backwards and forwards through the recursive evaluation of *(GCD 5 35)*. The strange identifier *lambda* is a placeholder for our function, GCD, at each step.

It may help your thinking to accept (without proof) the following mathematical facts: Fact 1: The GCD of two positive integers $n1$ and $n2$ is the same as the GCD of $n1$ and the remainder of $n2$ after dividing by $n1$. Fact 2: The GCD of any non-negative integer and zero is the given non-negative integer.

```
;; GCD : number number -> number
;; To find the Greatest Common Denominator (GCD) of two non-negative whole numbers.
;; example 1: (GCD 5 7) produces 1.
;; example 2: (GCD 15 35) produces 5.
(define (GCD n1 n2)
  (if (zero? n2) n1
      (GCD n2 (remainder n1 n2))))

(GCD 15 35)
```

SOLUTION: Once I got the definition typed, and started up the debugger, I felt a bit overwhelmed by information. I eventually got used to the pattern: the code on the left-hand side is highlighted in green, and it becomes (after Scheme gets through with it), the code highlighted in purple on the right-hand side. The lambda keyword is strange, but I mentally substituted "GCD", and it simply repeated the definition I had typed.

To begin with, n1 was replaced by 15 and n2 was replaced by 35 in the definition. Then (zero? n2) was replaced by false (since 35 is not zero), and in the next step we were on the second option of the if statement. This called GCD with values n2 (or 35) and (remainder n1 n2) (or (remainder 15 35), which is 15). There doesn't seem to be much progress being made here. However, after stepping a bit more, we have an expression to evaluate (GCD 15 5), so the numbers appear to be getting smaller. Stepping some more yields (GCD 5 0), and that's the one my definition knows how to handle, it returns 5.

I tried some other numbers, and I can give an approximate explanation. If the second number is 0, the GCD is the first number. Otherwise, we take the remainder after division by the second number as the new second number, and put the second number in the first numbers place, and call GCD again. This process of repeatedly taking remainders will surely reduce the second number to 0 eventually (I can't prove this, but it seems true), so a result will be produced. And the given fact says that this is the same GCD as we're looking for in the first place.

10. Please feel free to ask for help/hints from the TAs and instructors on this exercise, creating a definition for the function *make-list*. Keep working in the same definitions pane as your previous definitions, saving your work occasionally by clicking the *Save* button. Observations, conjectures, and explanations go in *journalA3*.

    The idea is to create a function that produces lists with n copies of some object. The natural way to carry out repetitive tasks in Scheme is with recursion, the technique from the previous exercise. The key idea in writing this short definition is in the following paragraph:

    If n is less than 1, produce an empty list. Otherwise append a list containing a single copy of the object to a list of $n - 1$ copies of the object.

You might want to re-read the above two sentences a few times as you work on writing the definition below. I believe it almost spells out the code you have to write to create the function, but it takes a lot of staring to get used to recursion.

```
;; make-list : number object -> list
;; To produce a list consisting of number copies of object.
;; example 0: (make-list 0 5) produces '()
;; example 1: (make-list 3 5) produces '(5 5 5)
```

SOLUTION: This took a few passes for me. I imitated the previous exercise, since I know what to do for the shortest possible output, a list of zero copies: *(define (make-list n1 thng) (if (zero? n1) '() ...))*. Again, I'll need to be a bit more concrete than the ... I need to create a list of n1 copies of thng by creating a smaller list (solving a smaller problem...). Okay, suppose I could create a list of n1-1 copies of thng, then I could simply add a list with 1 copy, so I replace ... with *(append (list thng) ...)*. Now I'm stuck on how to make a list of n-1 copies of thng — it doesn't seem as though I've made any progress. I ask the Prof., and he says something unhelpful about using recursion now... as if I could just type in *(make-list (- n1 1) thng)* at this point! Just to make it clear how unhelpful he's been, I do type that in and try it out ... it for some weird reason it works.

I trace out a few small with the stepper, and then with paper and pencil, and it becomes clear that it should work. Each time Scheme encounters *(make-list (- n1 1) thng)* the value of *(- n1 1)* gets smaller, so eventually the case where we're making 0 copies is reached, and everything works out. Just for the heck of it I changed the recursive call to *(make-list n1 thng)* and ... I'll get back to you, it doesn't seem to stop.

SNEEZE AND SENSIBILITY

Keep DrScheme open to the definitions pane *sneeze.scm*, which has the definitions you created, as you read the next few paragraphs.

Every winter respiratory infections sweep through our population, carried on a sneeze. Depending on how strong your resistance is, your neighbour's sneeze today will be your sneeze tomorrow.

I have built a simplified model of a sneeze epidemic. The population is represented as a list of zeros (representing healthy individuals) with a single 1 (an infected individual). As each day passes, a healthy individual with an infected individual neighbouring them (to the right or left) will become infected if their resistance is less than the virulence of the infection. Once infected, they remain sick (and infectious) for a number of days (which I call duration), at which point they recover and become immune for a number of days (which I call immunity).

Depending on the value of *virulence* (a number between 0 and 100), *immunity*, and *duration*, the course of the epidemic will be different. For each individual, *resistance* is randomly chosen, each day, to be a number between 0 and 100. If a person has a sick neighbour, and their *resistance* is less than virulence, they become sick.

Open a separate terminal where you can copy `~heap/pub/epidemic.scm` to your current (A3) directory. In DrScheme, open *epidemic.scm* using the *File/New* menu. You will now have one definitions pane for the definitions you created, and saved in *sneeze.scm*, and another definitions pane with *epidemic.scm*. You will need to set the language level to *Advanced Student* in the *Language/How to Design Programs* menu, in order for some of the commands I have put in *epidemic.scm* to be recognized.

In *epidemic.scm*, hunt for the comments (the passages beginning with semicolons) that correspond to each of the definitions you created. Cut-and-paste (or carefully retype) your definitions immediately below those passages — no semicolons in front of your definitions, please!. When you're done, click *Run*.

If you complete the Scheme code correctly, you will be prompted for values of *virulence*, *duration*, *immunity*, and *days*. Try starting with 10, 15, 5, and 100, respectively. If you're successful, a two-dimensional simulation of the sneeze epidemic runs each time you click the *Run* button. You will get different simulations if you change the values for *virulence*, *duration*, *immunity*, or *days*. Record your observations and conjectures in *journalA3*.

SOLUTION: Cutting and pasting is messy, but eventually I got the code transferred, hit the "Run" button, and immediately got some errors. I'd forgotten to change the language level to Advanced, so I fixed that and was prompted for "virulence", "duration", "immunity", and "days". The creepy green epidemic filled my screen.

I played with the same numbers for a while, and though I got different patterns, I usually got an epidemic that died out after spreading for a while. I tried increasing the virulence to 20 (the epidemic almost always continued for 100 days and spread to other individuals quickly). I tried increasing the duration to 30 days, and the epidemic almost reached 100 days, but was "skinnier" — it didn't branch out to new individuals as quickly as when I increased the virulence.

So far, this seems like a plausible simulation of a disease epidemic: if it's really virulent, it spreads very quickly from individual to individual. A less virulent but long-lasting disease spreads, but more slowly. The end effect is the same in both cases: lots of sick people. I also tried reducing the immunity period to zero, and the disease would re-infect an individual occasionally after just 1 day (nasty).

WHAT TO SUBMIT

Please submit the following files:

- *journalA3*
- *sneeze.scm*